Workflow Design using Fragment Composition (Crisis Management System Design through ADORE)

Sébastien Mosser, Mireille Blay-Fornarino, Robert France

▶ **To cite this version:**

Sébastien Mosser, Mireille Blay-Fornarino, Robert France. Workflow Design using Fragment Composition (Crisis Management System Design through ADORE). LNCS Transactions on Aspect-Oriented Software Development, 2010, Special issue on Aspect Oriented Modeling, pp.1-34. hal-00531026

HAL Id: hal-00531026
https://hal.science/hal-00531026

Submitted on 1 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Workflow Design using Fragment Composition
Crisis Management System Design through ADORE

Sébastien Mosser[1], Mireille Blay–Fornarino[1], and Robert France[2]

[1] University of Nice – Sophia Antipolis
CNRS, I3S Laboratory, MODALIS team
Sophia Antipolis, France
{mosser,blay}@polytech.unice.fr
[2] Computer Science Department
Colorado State University
Fort Collins, CO 80523-1873
france@cs.colostate.edu

Abstract. The Service Oriented Architecture (SOA) paradigm supports the assembly of atomic services to create applications that implement complex business processes. Assembly can be accomplished by service orchestrations defined by SOA architects. The ADORE method allows SOA architects to model complex orchestrations of services by composing models of smaller orchestrations called *orchestration fragments*. The ADORE method can also be used to weave fragments that address new concerns into existing application models. In this paper we illustrate how the ADORE method can be used to separate and compose process aspects in a SOA design of the Car Crash Crisis Management System. The paper also includes a discussion of the benefits and limitations of the ADORE method.

1 Introduction

In the Service Oriented Architecture (SOA) paradigm [1] an application is an assembly of services that typically implements a mission-critical business process. A design model of a SOA application describes how services will be orchestrated when implemented, and is typically created by business process specialists. A factor that contributes to the difficulty of developing complex SOA applications is the need to address non-orthogonal concerns, such as security, data persistence and fault tolerance, in business processes. The complexity of addressing non-orthogonal concerns during design can be effectively managed by using process modeling techniques that support separation and composition of non-orthogonal concerns. The use of these techniques allows developers to model orchestrations that weave and coordinate non-orthogonal behaviors across a set of services. Support for separately modeling and composing different aspects of a process can also make it easier to extend models of SOA applications with behaviors that address new or changed concerns, or with features that improve system properties such as response time.

Existing process modeling tools and formalisms (*e.g.* BPMN [2], BPEL [3]) do not provide the separation of concerns mechanisms needed to separate and compose non-orthogonal process aspects. The ADORE (*Activity moDel supOrting oRchestration Evolution*) method[3] supports a compositional approach to modeling complex orchestrations. Models describing smaller orchestrations of services are composed to produce a model describing an orchestration of a larger set of services. The models of smaller orchestrations, called orchestration *fragments*, describe different aspects of a complex business process, where each aspect addresses a concern. Aspects that address non-functional concerns may be non-orthogonal and thus orchestrations that involve weaving aspects at different points in one or more services may be required. ADORE allows business process specialists to model different process aspects separately and then compose them. The support for separation of concerns helps to tame the complexity of creating and evolving models of large business processes in which many functional and non-functional concerns must be addressed.

ADORE consists of a process modeling language and a composition algorithm. Composition is automated, and thus business process specialists do not need to manually compose the fragments they create. ADORE can be viewed as a method that integrates Model Driven Development and Aspect Oriented Modeling (AOM) techniques to support development of SOA applications. From an AOM perspective, the fragments are aspects and the ADORE composition mechanisms use join points, pointcuts, and advise to determine what, where and how to compose. It is important to note that ADORE provides support for modeling only business process concerns, that is, activities and their orchestrations. For example, the modeling of object structures manipulated by the activities is not supported in ADORE. In this respect, ADORE can be considered to be complementary to other AOM approaches that support modeling of object structures, but not of activities and their orchestrations.

In this paper we demonstrate how the ADORE method can be used to model processes in the Car Crash Crisis Management System (CCCMS). In Section 2 we give an overview of the ADORE modeling language and show how it can be used to model different aspects of the CCCMS as orchestration fragments. In Section 3 we describe the ADORE composition mechanism and illustrates its use on the CCCMS. Consistency checking of ADORE models is discussed in Section 4. Section 5 presents an evaluation of the approach with respect to its ability to reduce cognitive load and tedious, error-prone manual effort through automated support for composing aspects.. In Section 6 we give an overview of a prototype tool we developed to support the use of ADORE and discuss the limitations ofADORE method , and in Section 7 we discuss related AOM work. Section 8 concludes the paper by summarizing the results and outlining planned further work.

[3] See `http://www.adore-design.org`

2 Using ADORE to Model the CCCMS

In this section, we illustrate how the CCCMS can be modeled using the ADORE method. The complete set of models can be found on the following website:

`http://www.adore-design.org/doku/examples/cccms/start`

After a brief introduction to ADORE meta–model, we show how it can be used to realize textual use case descriptions of main success scenarios as orchestrations (cf. 2.2). A main success scenario describes a main process flow that has only one exit point [4]. Alternate scenarios and error handling are described as use case extension behavior, realized as fragments in our approach (*cf.* 2.3). Non-functional requirements are also realized as fragments (*cf.* 2.4).

2.1 The ADORE Method: An Overview

The BPEL is defined as *"a model and a grammar for describing the behavior of a business process based on interactions between the process and its partners"* [3]. It defines 9 different kinds of atomic activities (*e.g.*, service invocation, message reception and response) and 7 composite activities (*e.g.*, sequence, flow, if/then/else), plus additional mechanisms such as transactions and message persistence.

In ADORE, an orchestration of services is defined as a partially ordered set of activities, denoted as $\mathcal{A}^\star$. The different types of activities that can be defined in ADORE include *(i)* service invocation (denoted by `invoke`), *(ii)* variable assignment (`assign`), *(iii)* fault reporting (`throw`), *(iv)* message reception (`receive`), *(v)* response sending (`reply`), and *(vi)* the null activity, which is used for synchronization purpose (`nop`). In an ADORE process model, each process starts with a `receive` activity and ends with `reply` or `throw` activities. Consequently, the ADORE meta-model contains all the atomic activities defined in the BPEL normative document except the `wait` (stopwatch activity) and the `rethrow` (assimilated as a simple `throw`) activities.

ADORE defines four different types of relationships between activities:

wait : Start an activity a after the end of an activity a'.
guard : Describes a conditional *wait* relationship[4].
fail : Start an activity a when an activity a' throws a given error.
weak–wait : Represents a mutual exclusion relationship [5].

As the ADORE meta-model does not define composite activities, BPEL composite constructions are reified using the different relations available in the meta-model. A sequence of activities is defined by a `waitFor` relation; If/then/else

[4] an activity a guarded by $(a', v, true)$ will start after the end of a' if its output v is equal to $true$.

[5] *Weak–wait* is denoted as $\ll$. Weak relations like $\{a \ll c, b \ll c\}$ means that c will start after the end of the first activity picked from the set $\{a, b\}$. Note that wait relations like $\{a' \prec c', b' \prec c'\}$ means that c' will start at the end of a' and b'

flows are modeled using `guard` relations. Unlike BPEL which uses composite activities to implement loops, ADORE uses *iteration policies*. As loop handling in ADORE is out of the scope of this paper, the interested reader can find a full description of it in our previously published work [5]. A more complete description of the ADORE modeling language can be found on the project web site.

According to the ERCIM working group on software evolution [6], *aspect–oriented* approaches rely at a syntactic level on four elementary notions: *(i) joinpoints*, *(ii)*, *pointcuts (iii)*, *advice* and finally *(iv) aspects*.

Joinpoints represents the set of well-defined places in the program where additional behavior can be added. In the context of ADORE, we use activities to reify this notion. *Pointcuts* are usually defined as a set of joinpoints. In ADORE, one can identify sets of activities as pointcuts using explicit declarations (*e.g.*, use $\{act_3, act_4\}$ activities as pointcuts) or computed declarations (*e.g.*, all activities calling the service *srv*). *Advice* describes the additional business logic to be added in the initial system. ADORE represents systems as a set of business processes. We reify *advices* in an endogenous way as business processes called *fragment* (see section 2.3). Finally, *aspects* are defined as a set of pointcuts and advices. ADORE uses *composition directives* to bind fragments to set of activities.

An *aspect–oriented* approach also defines at least two mechanisms: *(i)* aspect weaving (to integrate aspect into base program) and *(ii)* aspect ordering (to define an order between aspect woven around the same join points). ADORE differs on the second point, as we define a *merge* algorithm to compose fragments around a *shared join point* [7] instead of ordering them (see section 3). Based on these definitions, we consider ADORE to be an aspect-oriented modeling approach.

2.2 CCCMS as Service Orchestrations

In this paper we describe a SOA design of the CCCMS. The use cases defined in the requirements document are treated as informal specifications of service orchestrations. We use the ADORE method to produce a design model of orchestrations. It is important to note that the focus is not on modeling the internal behavior of services or activities, but on the modeling of orchestrations. In the approach, a use case is realized as an orchestration of services. The main success scenario in a use case is realized as a base orchestration, while each use case extension is realized as an orchestration fragment. The fragments are composed with the base model to produce a model describing a realization of the use case as a service orchestration. We illustrate the ADORE method using the *"Capture Witness Report"* use case (`#2`). Below is the description of the main success scenario for this use case, extracted from the requirement document:

Coordinator *requests Witness to provide his identification.*

1. *Coordinator* provides witness information to *System* as reported by the witness.
2. *Coordinator* informs *System* of location and type of crisis as reported by the witness.
   In parallel to steps $2-4$*:*

2a.1 *System* contacts *PhoneCompany* to verify witness information.
2a.2 *PhoneCompany* sends address/phone information to *System*.
2a.3 *System* validates information received from the *PhoneCompany*.

3. *System* provides *Coordinator* with a crisis-focused checklist.
4. *Coordinator* provides crisis information to *System* as reported by the witness.
5. *System* assigns an initial emergency level to the crisis and sets the crisis status to active.

Use case ends in success.

The above scenario is realized as an orchestration of four service providers: *(i)* `cms` represents the services provided by the *System* actor, *(ii)* `phoneCie` represents the services provided by the *PhoneCompany* actor, *(iii)* `ui` represents services used to interact with the *Coordinator* actor and *(iv)* `msgBus` represents services used to broadcast the mission status. It only uses the *wait* relations between activities.

Step	Acts.	Step	Acts.	Step	Acts.	Step	Acts.
1	$\{a_{10}, a_{11}\}$	2	a_2	3	a_3	4	a_4
5	$\{a_{50}, a_{51}\}$	2a.1	$a2a_{12}$	2a.2	$a2a_{12}$	2a.3	$a2a_3$

Step $\leftrightarrow$ Activities correspondences

Fig. 1. Orchestration representation for the *Capture Witness Report* use case (`#2`)

Fig. 1 shows a graphical ADORE model of the realization. The first step in the use case scenario is realized by two activities:

$a10$: The $promptWitnessInfo$ operation provided by the *Coordinator* is invoked. This operation requests and records witness information.

$a11$: The *setWitness* operation provided by the `cms` is called to add the witness information to the current crisis.

Steps 2, 3 and 4 in the main use case scenario (getting crisis preliminary information, building a crisis–dedicated checklist and prompting for check list answers) are each realized by a single activity: a_2, a_3, a_4 respectively. Steps $2a.1$, $2a.2$ and $2a.3$ must be done in parallel with steps $2-4$. Steps $2a.1$, and $2a.2$ are realized by the activity a_2a_{12}, while step $2a.3$ is realized by a_2a_3. Finally, step 5 is realized by two activities in the ADORE model: a_{50}, which assigns the emergency level, and a_{51}, which sends a message on the message bus indicating that the status of this crisis is now *"active"*.

2.3 Realizing Use Case Extensions as Orchestrations Fragments

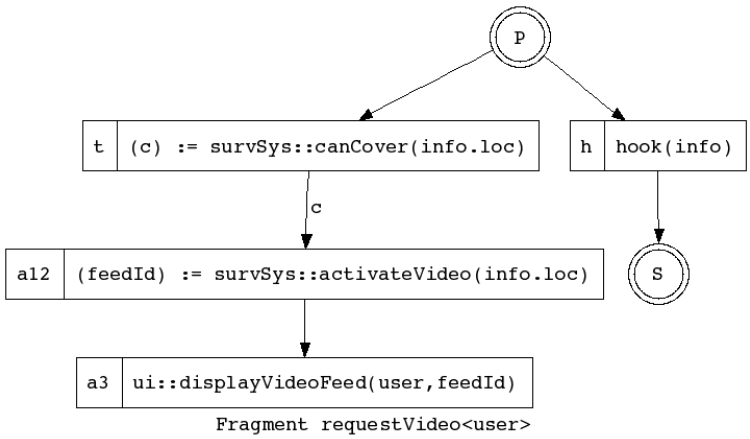
In ADORE, a *fragment* is a composable orchestration, that is, an orchestration that can be plugged into others. A fragment corresponds to a specific concern and use a partial point of view on its target. We reify this view using three special activities that are mandatory in each fragment. A special activity, called a *hook* (assimilated as a *proceed* in ASPECTJ [8]), represents where the fragment will be connected into an existing orchestration. An activity $\mathbb{P}$ represents *hook* predecessors, and $\mathbb{S}$ represents hook successors in an ADORE process structure.

A fragment can use *a-priori* unknown external entities. These entities are modeled as *fragment parameters*. At composition time these parameters are resolved by syntactically replacing them with references to defined entities. This parameterisation mechanism allows modelers to define generic fragments that can be instantiated and used in different contexts.

The *Capture Witness Report* (use case `#2`) describes six extensions to the main success scenario. We take as an example the extension `#3a`, stated as follows.

> *In parallel to steps* $3-4$*, if the crisis location is covered by camera surveillance:*
> 3a.1 *System* requests video feed from *SurveillanceSystem*.
> 3a.2 *SurveillanceSystem* starts sending video feed to *System*.
> 3a.3 *System* starts displaying video feed for *Coordinator*.

Figure 2 shows the fragment that realizes the above extension. The right branch of the fragment contains the hook, h, that represents the behavior in the targeted orchestration to which the fragment will be attached. A hook refers to a *block* of activities in the target orchestration. A block can consist of one or more activities. The hook predecessors ($\mathbb{P}$) are the immediate predecessors of the first activity in the target block, and the hook successors ($\mathbb{S}$) are the immediate successors of the last activity in the block. The right branch in FIG. 2 describes the following behavior: After the execution of the hook predecessors ($\mathbb{P}$), perform the activity block referred to by (h) and then continue with the hook successors ($\mathbb{S}$).



Step	Acts.	Step	Acts.	Step	Acts.
3a.1	a_{12}	3a.2	a_{12}	3a.3	a_3

Step ↔ Activities correspondences

Fig. 2. *RequestVideo* Fragment dealing with *Capture Witness Report* extension `#3a`

The left branch of the fragment represents the additional behavioral described in the use case the extension `#3a`. In parallel to the behavior described by the hook (h) the system first determines if the surveillance system can cover the crisis area (t). If this functionality is available[6] for this location, the process requests a video feed (steps 1 and 2 in the use case extension are aggregated in a single ADORE activity a_{12}) and then broadcasts it to the *Coordinator* interface (a_3). This fragment uses *guard* relations.

If we bind this fragment on the block of activities $\{a_3, a_4\}$ of the orchestration defined in figure 1, the following correspondences are automatically computed[7]:

$$\mathbb{P} \rightarrow \{a_2\}, \quad hook \rightarrow \{a_3, a_4\}, \quad info \rightarrow i, \quad \mathbb{S} \rightarrow \{a_{50}\}$$

2.4 Realizing Non–Functional Concerns as Fragments

ADORE can be used to model non–functional (NF) properties that can be realized as system behaviors. We realized three NF–concerns in the context of the CCCMS case study: *(i)* persistence, *(ii)* security and *(iii)* statistics logging. We define five fragments to implement these concerns in the CCCMS.

[6] The c label on the $t \xrightarrow{c} a_{12}$ arrow represents a guard: a_{12} will start only if `c` is evaluated as true after `t`'s execution

[7] Variable unification ($info \rightarrow i$) is based on type equivalence. One can explicit a given unification when such a mechanism fail (*e.g.*, two variable with the same type in the hook)

The *persistence* concern involved two fragments: `logCreate` (transforming a transient entity into a persistent one) and `logUpdate` (logging status information for a persistent entity). We make the choice to handle employees (both `cmsEmployee` and `worker` entities) as persistent resource. As a consequence, the `logCreate` (*resp.* `logUpdate`) fragment must be woven each time a service invocation creates (*resp.* uses) such an entity.

We realized a part of the *security* concerns through the following scenario: "an *idle* employee must be re–authenticated". We define a dedicated fragment (`authenticateWhenIdle`) to deal with this concerns, and weave it on all activities interacting with an employee through the user–interface.

We focus now on the *statistics logging* property description. The requirement document defines this property using the following text:

- *The system shall record the following statistical information on both on-going and resolved crises: rate of progression; average response time of rescue teams; individual response time of each rescue team; success rate of each rescue team; rate of casualties; success rate of missions.*
- *The system shall provide statistical analysis tools to analyze individual crisis data and data on multiple crises.*

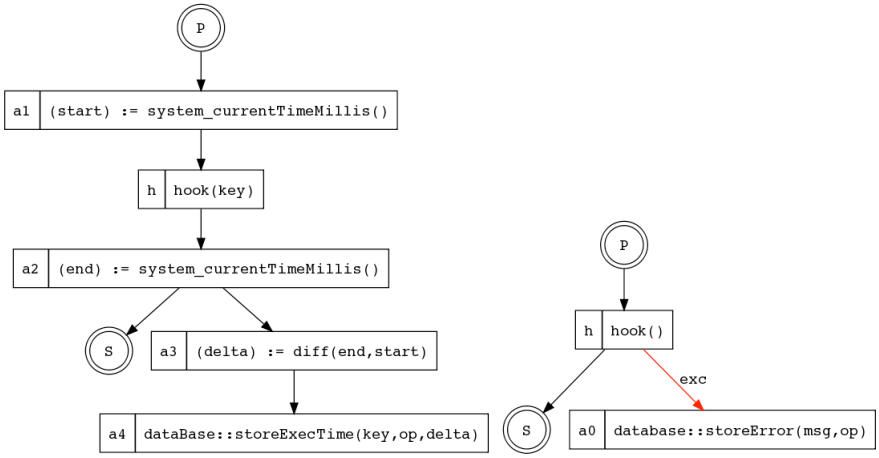
We realized this property by storing the execution time of a given action. We differentiate *normal* execution (the action succeeds, `logTime`) and *exceptional* execution (the action failed, `logError`). The two fragments realizing this property are depicted in fig 3.

The modeler can then decide to use one or both to track explicit activities in the CCCMS. The previously given definition is very close to the definition of business processes *performance indicators* (PI) given by CAROL [9]: *"A performance indicator can be defined as an item of information collected at regular intervals to track the performance of a system"*. As the business process management community identify explicit activities to be monitored by PI, it makes sense to use the same identification approach in the context of this case study.

We can also notice that the requirement document defines at the business scenario level a non–functional property. The informal specification requires in extensions `#1.1a` and `#3.1a` that a given CCCMS user must be authenticated! We naturally reified this concern as a dedicated fragment (`mustAuthenticate`), shared with the two associated business processes (`assignInternalResource` & `resolveCrisis`).

3 Composing Orchestrations

In this section we describe how the ADORE platform handles the composition of orchestrations, by describing the two algorithms involved in the process: *(i)* fragment weave and *(ii)* behavioral merge. We summarize in TAB. 1 the number of times each algorithm is called and the number of actions performed on the initial model to execute the composition.



(a) `logTime` Fragment (b) `logError` Fragment

Fig. 3. Non–functional fragments storing response times and errors

Modeled System	Fragment Weave	Behavioral Merge	Executed Actions
Business–driven CCCMS	23	5	2422
Including NF concerns	86	38	10838

Table 1. Algorithms usage (& associated actions) when modeling the CCCMS

3.1 Fragment Weaving

In this paper we give an informal description of the algorithm. A more formal and complete description can be found in our previously published work (see [10]).

The weave algorithm aims to integrate a given set of fragments into an orchestration. It does not rely on the order in which fragments are presented because it reasons about all fragments separately before finally performing the weaving actions on the target orchestration. But it can only weave a *single* fragment on each targeted activity. When several entities wust be woven at the same point, the behavioral merge algorithm (described in next section) must be used before weaving. As the weaving algorithm is endogenous, the resulting process is an ADORE orchestration that conforms to the ADORE meta-model.

The algorithm follows the following scenario:

1. *Determine where the fragment will be inserted in the base orchestration*: The modeler specifies the blocks of activities in the base orchestration that will be bound to the hooks in fragments. A mapping of a fragment hook to an activity block is called a *binding*. A *composition unit* is a set of bindings used to weave fragments into a base orchestration. A *binding* is formally defined as a tuple $\omega(f, a)$ where f is a fragment and a an activity or a set of activities.
2. *Determine the set of actions needed to compose fragments to the base orchestration*: For each binding, the composition algorithm computes the set of actions[8] that must be performed on the base orchestration to include the fragment.
3. *Perform the weaving*: When all actions are computed, the system executes the action set to produce the composed model.

To illustrate the principles, we use the *Capture Witness Report* use case (`#2`, cf. fig.1 p.5). It is realized in ADORE as fragments that realize the five extensions specified for this use case: `callDisconnected` (`#1a`, `#2a`), `requestVideo` (`#3a`), `ignoreDisconnection` (`#4a`), `fakeWitnessInfo` (`#5a`) and finally `fakeCrisisDetected` (`#5b`). These fragments must be composed with the `captureWitnessReport` orchestration that realizes the main scenario.

Figure 4 gives the composition unit (a set of bindings) defined by the CCCMS modeler that will be used to weave the fragments into the base orchestration. Each defined fragment is bound to a block of activities using an *apply* directive.

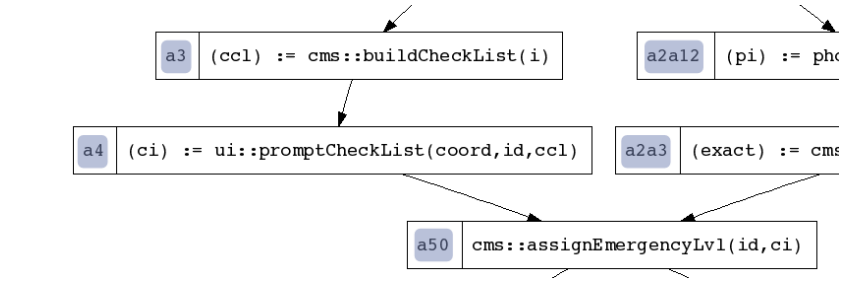
Fragment weave in ADORE is an endogenous process, and thus it is possible to weave a fragment into another fragment to produce a larger fragment. The last line of the composition unit weaves a fragment into another one. Such a composition is performed before using the targeted fragment into other weaving actions. In this case, it means that the `fakeCrisisDetected` fragment is woven into the `requestVideo` fragment. The resulting fragment is then woven into the targeted orchestration as required by the extension `#3a`.

[8] *e.g.* adding an activity, creating an order. All atomic actions defined over the ADORE meta-model are available on the project web site.

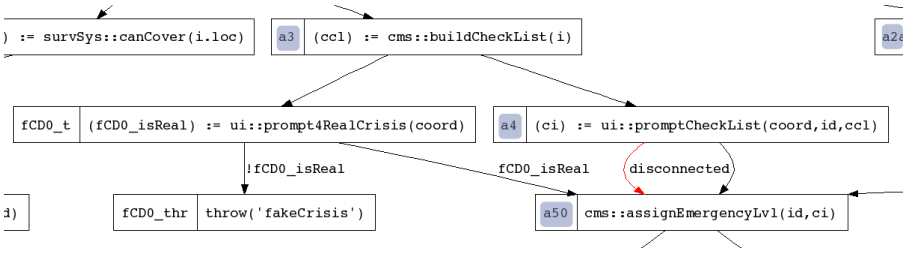
```
composition cms::captureWitnessReport {
  apply callDisconnected            => a10;     // 1a.
  apply callDisconnected            => a2;      // 2a.
  apply requestVideo(user: 'coord') => {a3,a4}; // 3a.
  apply ignoreDisconnection         => a4;      // 4a.
  apply fakeWitnessInfo             => a2a3;    // 5a.
  apply fakeCrisisDetected          => a4;      // 5b.
  apply fakeCrisisDetected => requestVideo::a3; // 5b.
}
```

Fig. 4. Composition unit to build the *Capture Witness Report* complete process

The orchestration model obtained for the complete use case after composition is really large, and represented in appendix[9]. We show only part of the obtained result in Fig. 5 to ameliorate the readability of the result. This figure shows the before/after composition views that focuses on activities $\{a_3, a_4, a_{50}\}$ of the `captureWitnessReport` orchestration.



(a) *Capture Witness Report* **before** behavioral composition (Extract)



(b) *Capture Witness Report* **after** behavioral composition (Extract)

Fig. 5. Illustrating the fragment weave process on *Capture Witness Report*

[9] The interested reader can browse **all** the composition results by visiting the case study web site. The figure 17 in annexe depicts the resulting orchestration after weaving all the fragments. But readability is difficult.

3.2 Behavioral Merge

When several fragments $\{f_1, \ldots, f_n\}$ must be woven into a process using the same *hook*, the algorithm automatically performs a preliminary *merge* of the fragment set to compute a *merged* fragment.

The *merge* algorithm performs a unification of the fragments' special activities ($\mathbb{P}$, $\mathbb{S}$ and *hook*) to build the merged fragment [10]. The merge algorithm relies on logical unification and substitution [11]. The process is then deterministic: for a given set of fragment, there will be only one possible merged result.

$$\{\omega(f_1, a), \ldots, \omega(f_n, a)\} \equiv \omega(merge(\{f_1, \ldots, f_n\}), a)$$

From the composition unit described in Fig. 4, the algorithm identifies that two fragments must be woven on the a_4 activity: `fakeCrisisDetected` (Fig. 6a) and `ignoreDisconnection`[10] (Fig. 6b). A merge of these two fragment is then required before weaving the merged fragment using a_4 as *hook*.

3.3 Pointcut Matching Mechanism

Adore does not focus on automatic pointcut matching as it is usual for business processer modeler to explicit the points they want to control in a given process.

However, the Adore meta–model intrinsically relies on set theory and first order logic. One can then use these formal tools to automate the pointcut matching phase of the system design.

We can take as an example the `logError` non–functional fragment, previously described. This fragment should be woven on activities which can potentially throw a fault. An activity a is a candidate for such a weaving since there exists another activity b linked to a by a $fail$ relation (on any φ fault). This statement can be formally defined using the following rule:

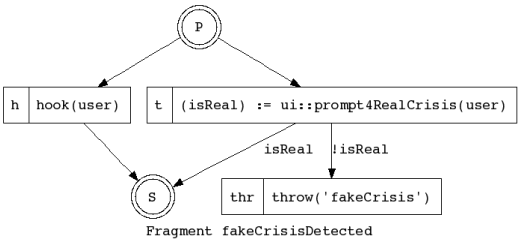
$$a \in \mathcal{A}^\star,\ \exists b \in \mathcal{A}^\star,\ \exists \varphi \in Faults,\ fail(a, \varphi) \prec b$$

4 Analyzing ADORE Models to identify inconsistencies

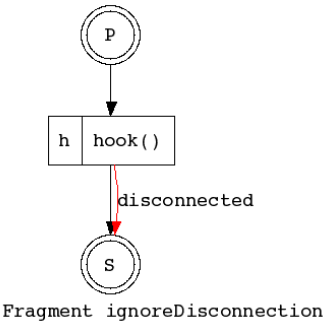
The large size and conceptual complexity of applications such as the Cccms is the main motivation for using aspect-oriented orchestration modeling approaches such as Adore. Separation of concerns reduces the complexity when focusing (locally) on one or a few fragments, but, at the same time, increases complexity when looking (globally) at the model as a whole. To tame this complexity we proposed to analyze the fragments and composed models to detect inconsistencies and bad-smells.

The Adore framework supports consistency checking at different phases. In the first phase (P_1), consistency checks are performed within each individual

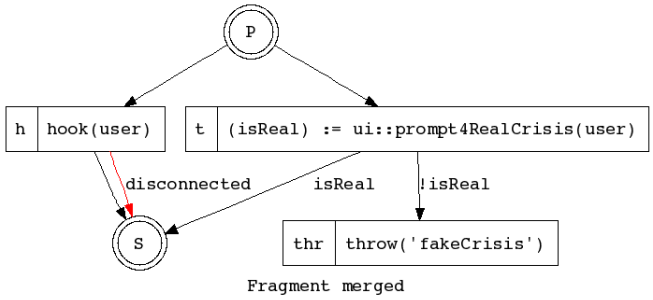
[10] The $h \overset{disconected}{\longrightarrow} \mathbb{S}$ red arrow represents the *catch* of an error thrown by h.



(a) 5b: $fakeCrisisDetected$



(b) 4a: $ignoreDisconnection$



(c) $merge(\{fakeCrisisDetected, ignoreDisconnection\})$

Fig. 6. Illustrating the fragment merge process through a simple example

fragment and orchestration model separately. In the second phase (P_2), consistency checking involves analyzing the set of matched joint points to discover symmetric composition of fragments. In the third phase (P_3) consistency checks involve checking the consistency of behavioral merge results[11]. The last consistency checks are performed on the final orchestration models (P_4).

Each check is described as a rule. A rule can be applied at different phases of the composition process. The result is interpreted differently according to the consistency checking phase. For instance, incompleness is interpreted as *bad smell* at phases P_{1-3} and as an error at phase P_4. Table 2 summarizes how ADORE interprets result of rule violations according to the phase.

Rule \ step of Checks	P_1&P_3	P_2	P_4
$\mathcal{R}_1$ Concurrent Ending	error	–	bad–smell
$\mathcal{R}_2$ Equivalent Calls	bad–smell	–	bad–smell
$\mathcal{R}_3$ Lack of Response	bad–smell	–	error
$\mathcal{R}_4$ Design Weakness	bad–smell	–	bad–smell
$\mathcal{R}_5$ Reflexive composition	–	error	–
$\mathcal{R}_6$ Recursive Condition	–	error	–
$\mathcal{R}_7$ Missing Weave	–	bad–smell	–
$\mathcal{R}_8$ Uninitialized Variable	error	–	bad–smell
$\mathcal{R}_9$ Invocation Cycle	bad–smell	–	bad–smell

Table 2. ADORE identification when detecting inconsistencies

We applied these rules on the CCCMS case study. Here we only present the rules that have helped to detect errors in our design of the application or that indicated deficiencies in the requirements. We classify these rules according to their relationships with the principles of aspect-oriented modeling.

Separation of Concerns ADORE focuses on separation of activities modeling according to non-functionnal aspects and extensions to main scenarios. A side effect of separating our concerns are concurrent response spawn, redundant invocations, ...

$\mathcal{R}_1$: No Concurrent Ending This rule checks whether an ADORE model is deterministic or not. A deterministic ADORE model has only one well-defined response activity for one path in the activity graph.

Such a situation must not exist in the base and fragment orchestrations. But concurrent responses can occur after composition, as a result of interactions

[11] This is an interesting property of the ADORE framework: the behavioral merge only produces new fragments at shared join points. As a consequence, the consistency check mechanism is only triggered on merged fragments, and thus avoiding combinatorial explosion of fragment combinations to check.

between fragments. Such a non-deterministic situation is considered as an error at phase P_1 and as a bad-smell at P_4. A design choice must be done by the modeler to fix it (*e.g.* introducing new activities, explicitly keeping the non–deterministic behavior).

In the *Capture Witness Report* use case (`#2`), main success scenario failures are defined in extensions. For example, scenario extensions `#5a` and `#5b` are defined *in parallel*. The first one defines a failure when the *Phone Company* actor cannot verify witness informations. The second one defines another failure when the *Coordinator* actor declares this crisis as a fake one. Since those two different situations are handled by two different actors in parallel, the system is non-deterministic at the requirements level.

We illustrate the situation in Fig. 7. Activity a_{131} represents failure defined in extension `#5a` and activity a_{139} represents the failure defined in extension `#5b`. As defined in the requirements document, there is no order between these two activities. As a consequence, the process behavior is non-deterministic when the two conditions are evaluated to false at the same time. In this particular case, as there is no error handling policy in the main orchestration (`resolveCrisis`), we decide to let the process be non-deterministic: the first encountered exception will be propagated to the caller without any more reasoning.

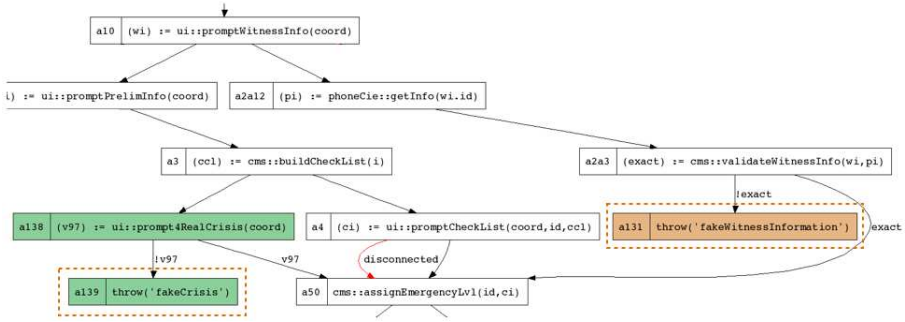


Fig. 7. $\mathcal{R}_1$ violation: Concurrent response spawn $\{a_{139}, a_{131}\}$ (extract)

This rule was particularly useful in detecting "bad" join points (*e.g.* some invocations to user interfaces with a `cmsEmployee` as parameter do not correspond to interactions that should be protected by authentication).

$\mathcal{R}_2$: No Unanticipated Equivalent Calls. Multiple fragments can introduce service invocations that are equivalent by weaving a same fragment on several activities or by requesting a same service in different fragments. A rule that brings these equivalent services to the attention of the modeler can cause the modeler to consider how the model can be refactored to avoid unnecessary redundant invocations of services.

To illustrate this rule, we focus on the *statistics logging* non–functional propertie. To record statistical information on response time of each rescue team, we weave the fragment `logTime` (cf. FIG. 3) around each invocation activity involving a `cmsEmployee`. When this fragment is woven on two immediately consecutive activities, it results in the situation depicted in FIG. 8 (extract of the *Execute Rescue Mission* use case). The activities a_{341} and a_{354} return the same time. The business process can be refactored by unifying these two activities. A knowledge is added in the system to optimize the orchestrations involving these useless concurrency calls (this pattern was detected three times in the CCCMS).

Another interesting illustration can be found in the *Capture Witness Report* use case. This rule has also detected that the *Coordinator* can inform the system that the crisis is a fake one by analyzing witness answers, *or* by looking at the video feed, if available. Consequently, the *Coordinator*'s approval will be requested twice when a video feed is available. In this case, we consider that the redundancy of the situation is handled by the `ui` service: it will not broadcast to the coordinator the same question multiple times.

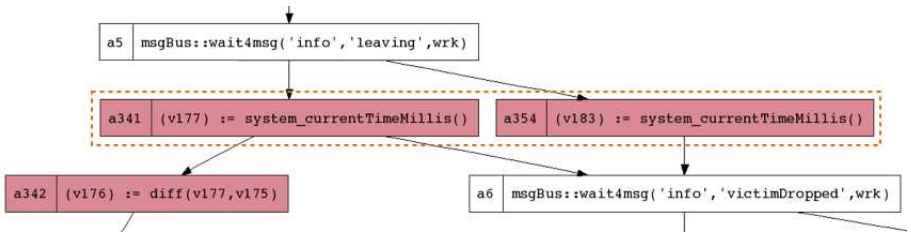


Fig. 8. $\mathcal{R}_2$ violation: *Unanticipated Equivalent Call* $\{a_{341}, a_{354}\}$

Model Incompleteness By nature fragments defined a partial point of view on existing entities. As a consequence, fragments do not necessarily have to be complete, *i.e.*, they only need to specify the variables and the activities that are relevant within the concern that is modeled.

$\mathcal{R}_3$*: Always a response.* This rule checks whether a path exists between an entry point (message reception or predecessors) and an exit point (response sending, error throwing or successors), under all possible branching conditions or error handling expressed inside the control–flow. An ADORE model that satisfies this rule is said to be complete with respect to paths from entry to exit points. At phase P_1 and P_3, a fragment or orchestration model addresses specific concerns and thus may not be a complete model; this happens when a path from an entry point to an exit point is not in the scope of the concern addressed by the model. One can reasonably expect though that the result of a composition is a complete

model and thus if the result of a composition violates this rule, the violation is classified as an error at phase P_4.

Use case `#6` (*Execute SuperObserver Mission*[12]) provides examples of incomplete models in phase 1. In the main success scenario, step 7 is defined as: *"7. System acknowledges the mission creation to SuperObserver."*. The following step is defined as: *"8. System informs SuperObserver that mission was completed successfully."*. In the base orchestration that realizes this use case, step 8 is done only if the mission creation has been acknowledged, but the textual use case does not define what happens if the mission is not acknowledged. FIG. 9a shows a partial view on the orchestration that realizes this main scenario (orchestration `cms::handleSupObsMission`). ADORE detects there is no path from a_7 to an exit activity when the guard is evaluated as $\neg ack$. The extension `#7a` is defined as a fragment represented in FIG. 9b. When this fragment is woven to `handleSupObsMission`, it automatically completes the process and makes it valid (as the composition result does not violate $\mathcal{R}_3$).

$\mathcal{R}_4$*: No Design Weakness.* This rule checks for common design weaknesses, specifically, unused variables, exceptions that are not caught, and call signature mismatches. These weaknesses are considered as *bad–smells*. Fixing such weaknesses is not mandatory for the modeler who can decide to ignore the violations.

The orchestration that we defined for use case `#1` (*"Resolve Crisis"*) provides examples of design weaknesses. It states that "*Resource* submits the final mission report to *System*". But this `report` is never used again in the use case. In the same use case, step 1 requests that the *Coordinator* capture witness report. This step refers to use case `#2` (*"CaptureWitnessReport"*), which can fail when a fake crisis is detected. The weakness is that there is no extension defining how the system should manage witness report when a fake crisis is detected (i.e., this exception is not caught).

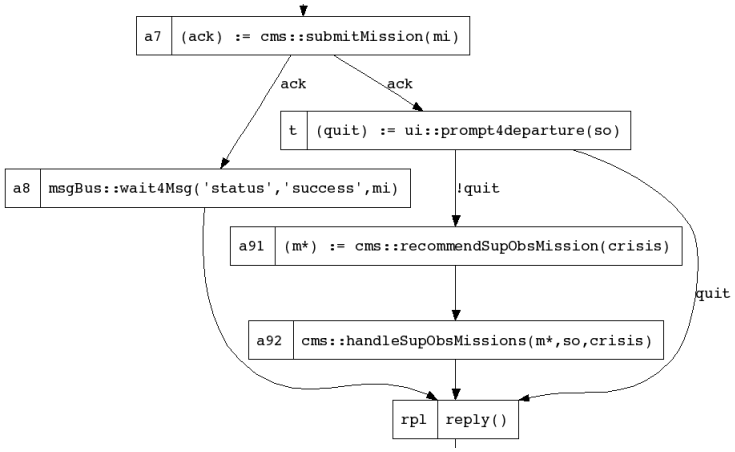
Weaving NF-fragments adds error throwing activities. But in the requirements document there is no information how to deal with these errors. This rule has identified this set of missing information in the requirements.

Pointcut interactions Other aspect-oriented approaches often use pattern matching mechanisms to identify join points. In our approach the modeler has to denote the join points explicitly or to use logical predicates to identify activities. ADORE can explicitly visualize and reason on this information to detect wrong matches and unintentional fragment compositions.

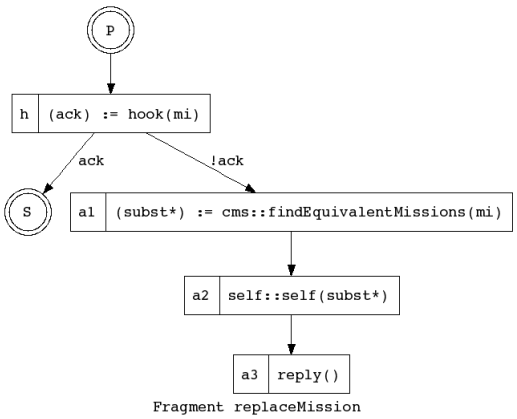
At phase P_2, the set of compositions directives is analyzed by the engine. We focus in this section on the fragment weaving directives, denoted as $\omega(f, a)$.

$\mathcal{R}_5$*: No reflexive composition* Pointcut matching can lead to try to weave a fragment on itself. We forbid this weaving as it does not make sense in ADORE.

[12] The intention of the *SuperObserver* actor is to observe the situation at the crisis site to be able to order appropriate missions



(a) `execSupObsMissions` inconsistent initial orchestration (Extract)



(b) Extension `7a` fix the inconsistency ($h \equiv a_7$)

Fig. 9. Illustrating the *Lack Of Response* ($\mathcal{R}_3$) inconsistency

This situation can be identified using the following predicate:

$$\forall \omega(f,a) \in Directives^{\star}, a \notin Activities(f)$$

If such a weaving directive is identified, a bad-smell warning is raised, and the directive is retracted from the directives set. This detection was useful when introducing the *persistence* concern through the `logUpdate` fragment. We first match all service invocations using a `cmsEmployee` variable, but such an invocation is defined inside `logUpdate` itself!

$\mathcal{R}_6$*: No recursive composition* This rule is an extension of the previous one. It identifies circular weaving of fragments, detected as non-convergent critical pairs. We define this rule as the transitive closure of the following predicate:

$$\forall \omega(f,a) \in Directives^{\star}, a \in Activities(f'), \nexists \omega(f',a'), a' \in Activities(f)$$

We met this difficulty by a bad definition of pointcuts for security and persistence. We just said that all invocations with input a variable of type `cmsEmployee` must be submitted to access control and be logged. Consequently, password entry should be logged and some log activities require authentication. We modified pointcuts as presented before to handle this issue.

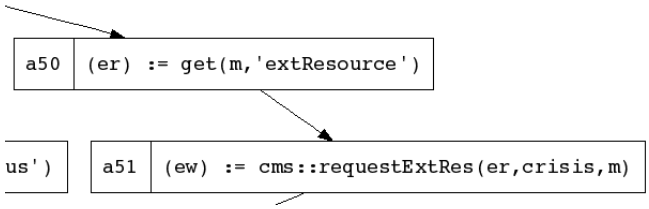
Fragments and Pointcuts interactions

R_7*: No missing weave.* When a fragment f is woven on an activity a' that is equivalent to another activity a that is not yet woven with f, ADORE displays a suggestion that the modeler may have forgotten to weave f on the activity a.

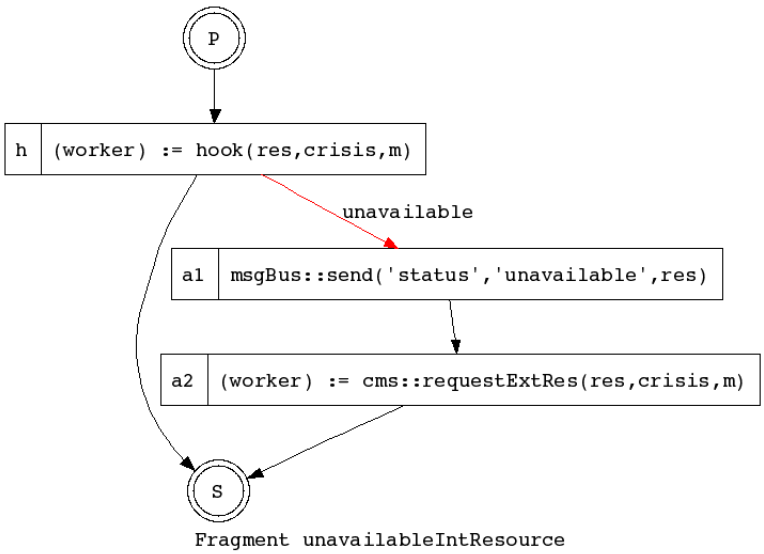
As an example, consider use case `#1` *"Resolve Crisis"*[13]. An extension (`#5a`) defines the process behavior when there is no available external resource. But extension `#4a` requests an external resource when there is no available internal resource. As a consequence, the error processing mechanism defined in extension `#5a` must also be triggered when extension `#4a` is used.

Another illustration can be found in use case `#2`. We weave fragment `unavailableIntResource` (FIG. 10a) into the base orchestration `handleAMission` (FIG. 10b). The ADORE platform detects that this fragment inserts an activity (a_2) which is equivalent to an initial activity (a_{51}). As a_{51} is used as a target for another fragment (`unavailableExtResource`, FIG. 11a), ADORE informs the modeler that she is *potentially* forgetting a composition directive. In this particular case, we decide to compose the two fragments to deal with the conflicting situation (no available resource at all).

[13] The intention of the *Coordinator* actor is to resolve a car crash crisis by asking employees and external workers to execute appropriate missions.



(a) Extract of `handleAMission` orchestration (UC `#1`)



(b) Ext. `4a`: `unavailableIntResource`

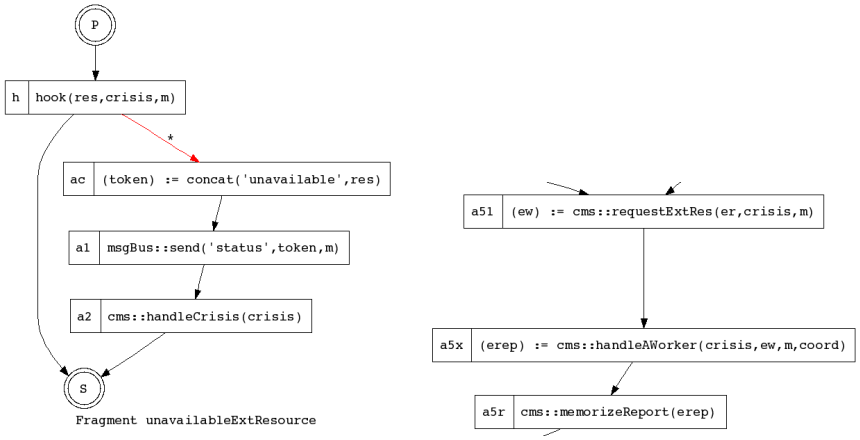
```
composition cms::handleAMission {
  apply unavailableIntResource  => a41;        // Ext 4a.
  apply unavailableExtResource  => {a51, a5x}; // Ext 5a.
}
```

Fig. 10. Illustrating the *Forgotten Weave directives* ($\mathcal{R}_7$): $a_2 \equiv a_{51}$

$\mathcal{R}_8$*: No Uninitialized Variable.* This rule checks whether a variable is used before being initialized. Ensuring this property before composition is not sufficient to ensure it after composition: Faulty interactions can lead to a composition result that violates this rule. Such a situation is typical when handling error in an orchestration. We define this property using the following predicate:

$$\forall a \in \mathcal{A}^\star, \forall v \in \mathit{Inputs}(a), \exists a' \in \mathcal{A}^\star, v \in \mathit{Outputs}(a') \wedge \mathit{path}(a' \rightarrow a)$$

Following the requirement document, in use case `#1`, each worker *must* submit a report to the system. But in extension `5a`, which describes how the lack of an external resource is handled, there is no information on how to deal with the missing report. We define the fragment `unavailableExtRes` (FIG. 11a) to represent the extension `#5a`. When the algorithm weaves this fragment with the targeted orchestration `cms::handleAMission` (FIG. 11b), the $\mathbb{S}$ activity is unified with a_{5r}. As a consequence (even if an error is thrown by a_{5X}) the process will try to memorize the expected report (a_{5r}), but this variable may not be initialized by a_{5X}. In this case, we decided to perform the initialization of the `erep` variable in the fragment.



(a) `UnavailableExtRes` fragment (b) `handleAMission` orchestration

Fig. 11. Illustrating the *Uninitialized variables* ($\mathcal{R}_8$) inconsistency

$\mathcal{R}_9$*: No Invocation Cycle* This rule checks whether fragments introduce a cycle of invocations. If there is no guarded activities in the execution path, we consider it as an error if not as a bad-smell. Several cycles of invocations were detected in the initial orchestrations and in the orchestration resulting from weaving.

This rule has detected an unexpected cycle of invocations in the first version of orchestrations integrating security. We prohibited the user to reconnect when

she remains idle a too long time. Modifying the associated pointcut modified the computed set of fragment weaving and consequently eliminated the cycle.

5 Quantitative Analysis

In this section, we highlight the need of separation of concerns when designing a large set of business processes. We collect quantitative data (obtained from ADORE) representing the initial system and then make a comparison with data collected after composition (with or without NF–fragments). We use a set of software metrics inspired by state-of-the-art research about business process quality [12]. The analysis we present in this section is based on data that is relevant only in the context of the AOM case study. The raw data for metrics computation are available on the case study web site, and the complete quantitative analysis is publicly available[14].

5.1 Coarse–Grained Structural Complexity: $|\mathcal{A}^\star|$

Definition: Software engineering community usually uses LOC (*"lines of code"*) to measure corse-grained software size. As the LOC metric is not directly suitable when dealing with business processes we translate this coarse grained complexity using the activity set cardinality (denoted as $|\mathcal{A}^\star|$).

Results: Our results according to this metric for the CCCMS case study are depicted in Fig. 12. We represent for each process the associated $|\mathcal{A}^\star|$, and compare three different versions of the CCCMS: *(i)* the initial system (main success scenario), *(ii)* the business–only system (including the scenarios extensions) and *(iii)* the final system (including both business extensions and non–functional concerns). The cardinality set average is multiplied by five between the initial system (7.83 activities in average) and the final one (39.25 activities in average).

Analysis: This chart clearly shows that the number of activities involved in the CCCMS realization grows really fast. Talking about the evolution cost, up to 10.000 elementary actions[15] need to be performed on the initial models to build the final ones. The composition algorithm takes in charge the complexity of building the complete process. In the next section, we focus on this induced complexity by looking at the provenance of entities inside computed processes.

5.2 Entity Provenance

Definition: This section is a corollary of the previous one. Based on the coarse–grained complexity of process, we identify how many activities came from the

[14] `http://spreadsheets.google.com/pub?key=tgS6qbzqo5CxTxlcTXbtsPA`

[15] *e.g.*, creating a variable or defining a new relation. A complete list of ADORE elementary actions is available on the tool website

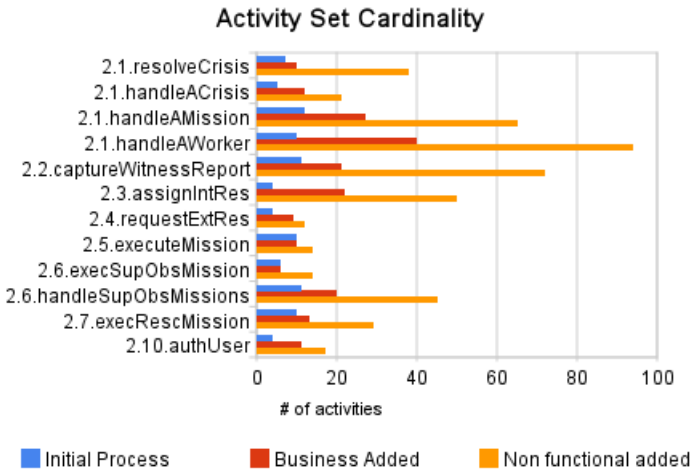


Fig. 12. Evolution of the $|\mathcal{A}^{\star}|$ indicator

initial orchestration, the business fragments and finally the non–functional fragments. This indicator allows us to quantitatively qualify which part of the final system was initially defined in the requirements. We normalized these values using the final cardinality ($|\mathcal{A}^{'\star}|$) to obtain an activity provenance percentage.

Results: Figure 13 represents these indicators for activities in the context of the CCCMS. We can immediatly notice than in average, more than 50% of a final process is defined as non–functional activities. Extrema values are interesting too: the `cms::assignIntRes` (use case `#3`) process contains only 8% of initial business activities. On the contrary the `cms::execRescMission` process (use case `#7`) is composed of more than 70% of initial activities.These values conform to the requirement documents, as scenario `#3` defines only two steps in the main scenario and nine in its extensions. On the contrary, scenario `#7` defines seven steps and only three in its extension.

Analysis: These indicators enforce the straightforward mapping between textual use cases and designed process. A large process with small extensions will be defined as a large orchestration and few fragments. But it also demonstrates the need of automatic composition, as in some case up to 75% of a process is defined as extensions of a main scenario.

5.3 Cognitive Load: Process Surface & Labyrinthine Complexity

Definition: The cognitive load indicator aims to quantify the intrinsic complexity of a business process. It is defined as an coarse grained approximation of

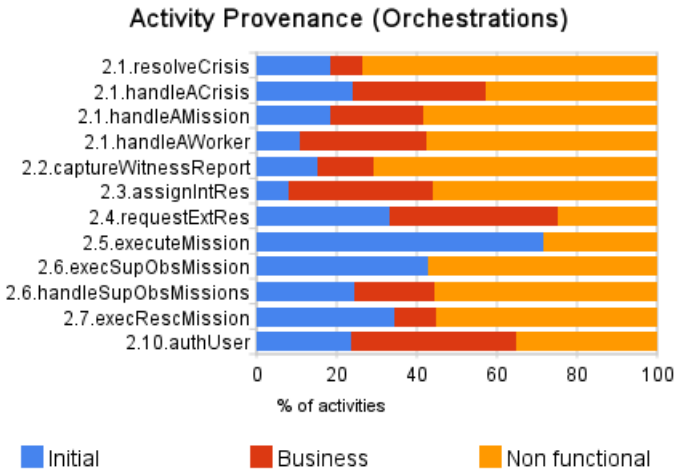


Fig. 13. Activities provenance in the final CCCMS

the Control Flow Complexity indicator [13], based on two simple concepts: the process *surface* and the *labyrinthine* complexity. *Surface* is computed as a product between process *width* (*i.e.* number of activities executed in parallel) and process *height* (*i.e.* longest path between an entry point and a exit point). The *labyrinthine* complexity represents the number of different path available in the process. Inspired by [14] who defines cognitive load of programs as a linear function (based on structural complexity), we define the cognitive load of an ADORE process as the multiplication of its surface and its labyrinthine complexity, normalized by the number of activities inside the process ($|\mathcal{A}^\star|$). This indicator allows us to apprehend both *computational* and *psychological* complexities as defined by CARDOSO *et al* [15].

$$CognitiveLoad(p) = \frac{Surface(p) \times Labyrinthine(p)}{|\mathcal{A}_p^\star|}$$

Results: We focus in this part on the business–driven CCCMS, *i.e.* the initial system and its business extensions[16]. For each process (excepting the `handleAWorker` one, as its cognitive load is 247.5), Figure 14 represents the sum of initial process and used fragments cognitive load, and the cognitive load of the final

[16] We consider than non–functional concerns should not be handled manually in an AOM approach

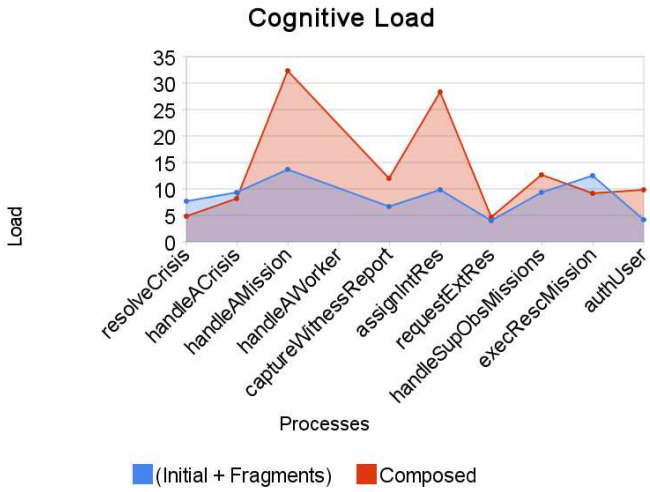


Fig. 14. Cognitive load indicator (Main Success Scenario + Business Extensions)

process. For five processes (50%), the final load is clearly higher than the sum of initial process and used fragments loads. For the other processes, the final cognitive load follows the same magnitude than the cumulated one.

Analysis: This indicator clearly illustrates the immediate advantage of separation of concerns to tame the complexity of designed artifacts. But it also highlights one of the weakness of the ADORE platform (and more generally the AOM approach). Designing small processes (or process without multiples extensions) using the separation of concerns paradigm may introduce an overload in the design process. This overload is not visible in terms of result, but can be summarized in the following sentence: *"When should one write several small concerns and then express a composition when he/she can directly express the expected result ?"*. Such a typical useless composition is illustrated in use case *Execute Rescue Mission* (#7), as the main scenario is implicitly designed to handle the sole extension defined on it: a request is sent in the main scenario, and the response to this request is handled inside the extension.

6 Implementation & Approach Limitations

In this section, we briefly describe the current implementation of the ADORE platform. We also describe some intrinsic limitations of the approach.

6.1 Tool Support

ADORE user interface is implemented as an EMACS major mode, as shown in Fig. 15. This mode hides in a user–friendly way the set of shell scripts used to interact with the underlying engine. The concrete ADORE engine is implemented as a set of logical predicates, using the PROLOG language. Rules described in section 4 are also implemented as PROLOG predicates.

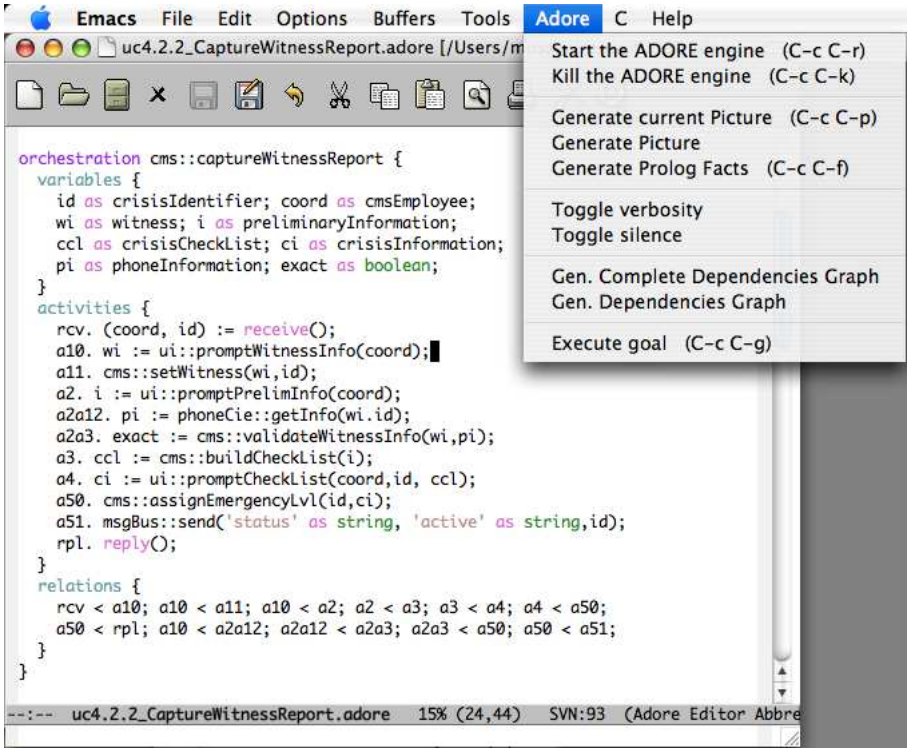


Fig. 15. ADORE editor, as an EMACS major mode

A dedicated compiler (defined using ANTLR[17]) implements an automatic transformation between ADORE concrete syntax and the associated PROLOG facts used internally by the engine (Fig. 16). As visualizing processes is important in design phase, ADORE provides a transformation from its internal facts model to a GRAPHVIZ[18] code which can then be compiled into a PNG file. It produces as a result a graphical representation of ADORE models, as depicted in all the figures showing ADORE models (*e.g.*, FIG. 6, FIG. 3).

[17] http://www.antlr.org/
[18] http://www.graphviz.org/

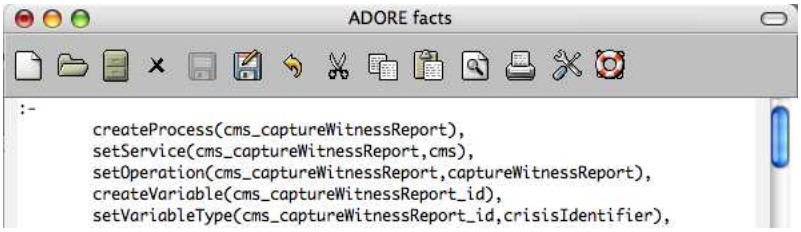


Fig. 16. PROLOG facts, generated by the ADORE compiler

Raw data (*e.g.*, number of activities, relations, process width) can be extracted as a XML document. This document can then be processed (manually or using technology like XSLT) to produce *before/after* graphics and benchmark the approach, as shown in section 5.

6.2 Composition Algorithms

The ADORE surface language allows modelers to define composition units. These compositions are compiled as PROLOG facts and analyzed by the engine. When the modeler asks ADORE to run the compositions, it determines for a given composition the different algorithms (*i.e.* fragment weave, behavioral merge) that need to be triggered and execute it.

We consider as an example a subpart of the `captureWitnessReport` composition (depicted in FIG. 4), which illustrate all the existing algorithms implemented in ADORE. We focus here on the a_4 activity, where the modeler asks to weave two different fragments (`ignoreDisconnection` & `fakeCrisisDetected`):

```
composition cms::captureWitnessReport {
  apply ignoreDisconnection => a4;
  apply fakeCrisisDetected  => a4;
}
```

To perform such a composition, the logical engine will execute a sequence of actions, represented in LISTING 1.1. It starts by *cloning* the two initial fragments into temporary entities (lines 1 & 2). The identification of a shared join point triggers the *merge* of the two involved fragments into a new one (line 3). Finally, this merged fragment is *woven* on the initial process (line 4), and a graph simplification algorithm is triggered (line 5) to make the result more understandable by humans (retracting useless relations).

```
1  doClone(ignoreDisconnection, tmp_1),
   doClone(fakeCrisisDetected, tmp_2),
3  doMerge([tmp_1,tmp_2], merged_1),
   doWeave([weave(merged_1, [cms_captureWitnessReport_a4])]),
5  doProcessSimplification(cms_captureWitnessReport),
```

Listing 1.1. Internal algorithms usage (automatically computed)

6.3 Pointcuts

ADORE does not provides any surface mechanisms to deal with pointcuts expressiveness. One can starts the engine in interactive mode and directly use PROLOG to query the model using logical unification. Losing the surface language implies that the modeler know the PROLOG programming language. But the immediate benefits are the unbounded possibilities of pointcut description offered by PROLOG.

We consider as an example the pointcut associated to the `logUpdate` fragment. This fragment must be applied on services invocation which use a variable of type `cmsEmployee` or `worker` as input. Based on the ADORE formal model, such a pointcut can be expressed as the following logical rule ($a \in \mathcal{A}^\star$):

$$Kind(a) = invoke \wedge \exists v \in InputVars(a), Type(v) = (\texttt{cmsEmployee} \vee \texttt{worker})$$

Using PROLOG, it is very easy to implement such a rule, as shown in LISTING 1.2. We provide on the website a PROLOG *toolbox*[19] to automate recurring patterns identification.

```
cut4logUpdate(Acts) :-
  findall([A], ( findVarUsageByType(cmsEmployee,in,A),
                 hasForKind(A,invoke)), CmsEmployees),
  findall([A], ( findVarUsageByType(worker,in,A),
                 hasForKind(A,invoke)), Workers),
  merge(CmsEmployees, Workers, Raws), sort(Raws, Acts).
```

Listing 1.2. PROLOG code associated to the `logUpdate` non–functional concern

We provide in ADORE a meta–predicate (LISTING 1.3) used to automate the pointcut mechanism. This predicate calls the user-defined pointcut predicates and then automatically build the associated *weave* directives.

```
build(FragmentName, PointcutPredicate, Directives) :-
  call(PointcutPredicate, List),
  findall(weave(FragmentName,E), member(E,List), Directives).
```

Listing 1.3. ADORE Meta–predicate used to automate pointcut matching

One can then define a pointcut as a unary predicate and asks ADORE to build the composition directives associated to it, as shown in (LISTING 1.4).

```
?- build(logUpdate, cut4logUpdate, L), length(L, Count).
L = [weave(logUpdate, [a13]), weave(logUpdate, [a20]),|...],
Count = 72.
```

Listing 1.4. Computing composition directives associated to `logUpdate`

[19] such as the `findVarUsageByType` predicate which identify activities that use a specific type in their associated variables.

6.4 Approach intrinsic limitations

Abstraction at workflow level Our approach aims to support aspect–oriented modeling at the level of workflows. The weaving of aspects inside an activity is not managed by the ADORE framework. In other words, we consider atomic activities and service as black boxes and do not provide any mechanisms to enhance them internally. Moreover, when designing such a big system as CCCMS, other AOM features are needed to design structural information about services and data. This is another limitation of the ADORE platform and one of our perspectives.

Scalability The section 5 demonstrated the scalability of the approach in the CCCMS context. The possibility to vizualise and analyse partial composition such as behavioral merge results help taming the complexity of business processes design. One of the limitation of ADORE is to provide visualizations only for separate entities: when a system involves many processes, it is necessary to have a holistic point of view on compositions and business processes to grasp it. So other visualization methods are needed to tackle complexity of compositions at the global system level.

Incremental composition ADORE works on set of fragments and directives to build final models. The complete approach relies on the existence of all the needed artifacts during compositions. To support incremental composition, modelers add new composition directives and then run the composition algorithms on this new directives. There is no tool support to let the modeler customize manually the final process and keep trace of such actions. These actions will be lost when re–running the algorithms.

Pointcut abstraction level ADORE framework does not support complex pointcut definition using a surface language. As soon as we need quantification we need to use directly the logical back–end. This approach is powerful and supports almost all kind of quantification. But the cost is a definitive lost of abstraction mechanisms. We do not consider it as an insurmountable drawback since business process modeler normally use explicit targets when they use process indicators.

Weaving and fragment instanciation The ADORE engine relies on the PROLOG unification to binds hook variables with real ones. However when the algorithm cannot unify the variable by itself, modeler must designate the substitutions between variables. This can be a complex task if unified services have a big set of parameters.

Reflexivity There is no real reflective support in ADORE. It only allows the usage of a `self` keyword to represent the current orchestration. We counterbalance this lack of reflexivity by using fragment parameters. This is not a definitive solution, and integrating such concern in the ADORE meta–model is an ongoing work.

7 Related Work

SOA and AOP Several approaches fill the gap between orchestrations and AOP (*e.g.*, [16], [17],[18]). These approaches rely on the BPEL language and impose to use dedicated BPEL execution engines to interpret the aspects. ADORE preaches technological independence and exposes itself as a *model* to support composition [10]. Instead of interpreting *aspectized* BPEL code, ADORE focuses on design of workflows by composition and weaving of fragments. When workflow designs are complete, we aim to generate complete orchestrations of services, executable in any industrial engine. This step of transformation is managed by means of correspondences between service models and web services urls. This transformation was used in the national project (FAROS consortium) to automatically integrate contacts in an orchestration[20].

SOA and AOM Many modeling languages (e.g., UML profiles and domain-specific languages) have been proposed to specify non-functional properties in SOA. In [20] the authors propose a MDD framework consisting of (1) a UML profile to graphically specify and maintain non-functional aspects, and (2) a MDD tool to transform profiled UML model to application code. The semantic of NF-properties is managed by model to code transformations. Reasoning on compositions of NF-properties is then managed at profile definition and transformation level. In [21], they extend this work to non-functional properties in business process models and propose an aspect oriented language to weave these properties inside a BPMN model. They do not address behavioral composition. To our knowledge, no study have proposed behavioral composition of business processes at business model level, but in sequence or state diagrams [22, 23].

Architectures and AOSD Architecture Description Languages (ADL) domain supports formal notations in order to define structure and behavior of software architectures. For instance in [24] authors use OCL to define invariant properties for component architectures and FSP Language to specify external component behaviors. This framework allows to partially check the architecture type consistency at structural and behavioral level. However these approaches imply to use several formalisms, and the result of composing specifications is not easy to understand. Such works could be a target for ADORE to insure properties such as termination. In [25], the authors manage the integration, in component assemblies, of new concerns represented as architectural aspects. This work relies on transformation rules describing precisely how to weave an architectural aspect in an assembly. Transformation engine manages the composition of the aspects. The behavioral modifications are then the consequences of architecture changes. This differs from our approach in that we focus on behavioral composition and keep the same formalism to express advices and base models. In [26],

[20] A technical report in French explains the transformation process [19]. A video demonstration of the transformation chain is also available on the project website: `http://www.adore-design.org/doku/examples/faros/start`

at component level, the authors express a similar point of view defining aspect components as encapsulation of advice codes.

Pointcut specifications The definition of pointcut languages play a central role in aspect-oriented approaches. But we did not directly contribute to this issue. In this case study we used PROLOG unification to match join points. The approach is really powerful but a little bit biased as pointcuts are defined at code level. However when an activity is identified as a join point, we use PROLOG unification, improved with type checking and knowledges, to unify fragment parameters. In this case study, adding semantic markers (*e.g.*, non-functional, business, toBeMonitored) [27, 28] and definition of dependent advices [29] would have simplified the expression of pointcuts. These definitions can then rely on abstractions such as: "business fragments are weaved with other business fragments only if they have to be monitored" or "*update* fragments on resource r requires the usage of the *create* fragment on the same resource".

Shared Join Points One of the strength of ADORE is to focus on the so–called *Shared Join Points* (SJP[7]) interactions spawn. When a set of fragments must be weaved at a same bloc of activities, the merging algorithm ensures the same result independently of the fragment order. The result of a merging can be visualized and analyzed using the set of rules presented in section 4. When interactions are detected, the modeler will enter knowledge at a fine–grained level (where coarse–grained is fragment re–ordering) to solve the conflict and then ease the interaction.

Fragment dependencies Aspect dependencies is defined in [30] as : *"one aspect explicitly needs another aspect"*. According to this definition, ADORE does not support fragment dependencies; We do not offer operators to define such relations between fragments, whereas work such as RAM [31] supports explicit expression of dependencies between aspects. In this case study, we needed to register the creation of resources to be able to trace resource state changes. The pointcuts for these two fragments determine complementary sets of join points, but we do not expose special operators to deal with these dependencies.

If we reread this definition in terms of *"an aspect that requires that the referenced aspects are woven previously"* [32], ADORE supports dependencies throughout weaving on fragments. Aspects weaved on another aspect reinforce this last one [30]. However these aspects are not really dependent as each one also works correct in isolation. We used several times weaving on fragments as several non-functional fragments crosscut business fragments.

Fragment Interactions Fragments as well as aspects can interact in multiple ways [30]. In section 4, we presented rules that detect unexpected interactions leading to unpredictable, duplicated or undesirable behavior but neither mutual exclusions nor conflicts among fragments. However when determining joint points

to require user to re-authenticate when he remains idle for thirty minutes, we first weaved authentication itself. Consequently we had introduced a cycle that has been detected by our detection rules.

Multi-view modeling In [31], authors propose RAM, an aspect-oriented modeling approach that provides multi-view modeling. We only support with ADORE behavioral point of view. However the consistency of the design will be greatly improved extracting service interfaces from orchestration and fragment definition and applying composition algorithm such as KOMPOSE [33] to obtain the service models (using UML class diagrams for example).

Workflow Modeling and Simulation Inspired by grid–computing community, ADORE proposes an algorithm (fully described in [34]) to automatically enhance a process with *set* concerns. Considering a process p handling a scalar data d, the algorithm can automatically transform p into a process handling a set of data $d^\star \equiv \{d1, \ldots, n_n\}$. Some workflow engines such as MOTEUR[35] support simulation. We are working on transformation from ADORE representation to GWENDIA language[36] to simulate the models resulting from the design, in order to analyze execution traces.

Visualization & Assessment Visualization of compositions is a very interesting problem when talking about composition assessment [37]. As ADORE allows users or programs to extract information from its internal representation, it is possible to extract process metrics and structural information from the engine. We are currently linking ADORE with MONDRIAN (an agile visualization tool [38]) to allow modeler to graphically visualize and asses their composition from an holistic point of view.

8 Conclusions & Further Work

This paper describes how to use the ADORE method to model the Crisis Management System Case Study as an SOA. In the approach, use cases are modeled as service orchestrations. For each use case, the main scenario is realized as a base orchestration, and the use case extensions and non–functional properties are modeled as fragments.

The ADORE framework supports composition mechanisms to automatically weave fragments into the base model. It also offers consistency checks to identify problematic *interactions* between fragments and base models.

A quantitative analysis of the approach has been performed based on a set of metrics inspired by state–of–the–art research on measuring business process complexity. It demonstrates the benefits of separation of concerns to tame the complexity of creating and evolving models of large business processes in which many functional and non-functional concerns must be addressed. It also established that automatic merging and weaving of fragments are essential to deal

with this complexity. The case study provides some evidence that the ADORE method is scalable. The orchestrations produced by the compositions in the case study are moderately large and complex as indicated by the size metrics we've gathered. Introducing non–functional concerns complexify the processes in a really deep way. The automated composition shields the developer from the complexity of composing orchestrations.

From a utility standpoint, we were able to realize all the use cases without encountering any methodological problems. ADORE intrinsically supports evolution of base orchestrations. We illustrate this feature in the context of this case study by considering each scenario extension as an evolution of the associated main success scenario. The introduction of non–functional properties that impacts processes behaviors such as persistence or statistical logging follows the same methodology.

An important ADORE goal is to support end–user design of orchestrations, where an end–user is a business process modeler. As a consequence, the modeling language utilizes concepts and expressions that should be familiar to business process modelers. Once a process modeler has specified where fragments are to be woven into the base model, ADORE composes the fragments and base models automatically and provides indications of possible problems arising out of interactions between fragments and base orchestrations. Some rules only identify design *bad–smells*, but others detect errors that make the composed result inconsistent. Such inconsistencies are detected and exposed to the process modeler, who can then fix it. As a consequence, we do not ensure an *a–priori* orchestration correctness property. It is interesting to notice that these rules were able to detect requirement weaknesses in the context of this case study.

One of the limitations of ADORE is that it does not support unweaving of fragments. Such unweaving is needed to support exploratory design of orchestrations. We plan to investigate this area (and others discussed in the limitation section) in our future work.

References

1. MacKenzie, M., Laskey, K., McCabe, F., Brown, P., Metz, R.: Reference Model for Service Oriented Architecture 1.0. Technical Report wd-soa-rm-cd1, OASIS (February 2006)
2. White, S.A.: Business Process Modeling Notation (BPMN). IBM Corp. (May 2006)
3. Jordan, D., Evedmon, J., Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., Guízar, A., Kartha, N., Liu, K., Khalaf, R., Konig, D., Marin, M., Mehta, V., Thatte, S., Van der Rijn, D., Yendluri, P., Yiu, A.: Web services business process execution language version 2.0. Technical report, OASIS (2007)
4. Mustafiz, S., Kienzle, J.: Drep: A requirements engineering process for dependable reactive systems. (2009) 220–250
5. Mosser, S., Blay-Fornarino, M., Montagnat, J.: Orchestration Evolution Following Dataflow Concepts: Introducing Unanticipated Loops Inside a Legacy Work-

flow. In: International Conference on Internet and Web Applications and Services (ICIW) AR=28%, Venice, Italy, IEEE Computer Society (May 2009)
6. on Software Evolution, E.W.G.: Terminology. Technical report, ERCIM (2010)
7. Nagy, I., Bergmans, L., Aksit, M.: Composing Aspects at Shared Join Points. In Hirschfeld, R., Kowalczyk, R., Polze, A., Weske, M., eds.: NODe/GSEM. Volume 69 of LNI., GI (2005) 19–38
8. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectj. In: ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming, London, UK, Springer-Verlag (2001) 327–353
9. : Performance indicators. In: BERA Dialogues. (1990)
10. Mosser, S., Blay-Fornarino, M., Riveill, M.: Web Services Orchestration Evolution : A Merge Process For Behavioral Evolution. In: 2nd European Conference on Software Architecture (ECSA'08) AR=14%, Paphos, Cyprus, Springer LNCS (September 2008)
11. Stickel, M.E.: A unification algorithm for associative-commutative functions. J. ACM **28**(3) (1981) 423–434
12. Vanderfesten, I., Cardoso, J., Mendling, J., Reijers, H.A., Van Der Aalst, W.M.: Quality Metrics for Business Process Models. BPM and Workflow Handbook (2007) 179–190
13. Cardoso, J.: Evaluating the process control-flow complexity measure. In: ICWS, IEEE Computer Society (2005) 803–804
14. Laue, R., Gruhn, V.: Complexity Metrics for Business Process Models. In Abramowicz, W., Mayr, H.C., eds.: BIS. Volume 85 of LNI., GI (2006) 1–12
15. Cardoso, J., Mendling, J., Neumann, G., Reijers, H.A.: A discourse on complexity of process models. In Eder, J., Dustdar, S., eds.: Business Process Management Workshops. Volume 4103 of Lecture Notes in Computer Science., Springer (2006) 117–128
16. Charfi, A., Mezini, M.: Aspect-oriented web service composition with ao4bpel. In: ECOWS. Volume 3250 of LNCS., Springer (2004) 168–182
17. Courbis, C., Finkelstein, A.: Weaving aspects into web service orchestrations. In: ICWS, IEEE Computer Society (2005) 219–226
18. Verheecke, B., Vanderperren, W., Jonckers, V.: Unraveling crosscutting concerns in web services middleware. IEEE Software **23**(1) (2006) 42–50
19. Blay-Fornarino, M., Ferry, N., Mosser, S., Lavirotte, S., Tigli, J.Y.: Démonstrateur de l?application SEDUITE. Technical Report F.4.4, RNTL FAROS (September 2009)
20. Wada, H., Suzuki, J., Oba, K.: A model-driven development framework for non-functional aspects in service oriented architecture. Int. J. Web Service Res. **5**(4) (2008) 1–31
21. Wada, H., Suzuki, J., Oba, K.: Early aspects for non-functional properties in service oriented business processes. In: SERVICES '08: Proceedings of the 2008 IEEE Congress on Services - Part I, Washington, DC, USA, IEEE Computer Society (2008) 231–238
22. Klein, J., Fleurey, F., Jzquel, J.M.: Weaving multiple aspects in sequence diagrams. Transactions on Aspect-Oriented Software Development (TAOSD) **LNCS 4620** (2007) 167–199
23. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P.: Matching and merging of statecharts specifications. In: ICSE '07: Proceedings of the 29th international conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2007) 54–64

24. Barais, O., Duchien, L. In: SafArchie Studio: An ArgoUML extension to build Safe Architectures. Springer (2005) 85–100 ISBN: 0-387-24589-8.
25. Barais, O., Lawall, J., Meur, A.F.L., Duchien, L. In: Software Architecture Evolution. Springer Verlag (2008) 233–262
26. Pessemier, N., Seinturier, L., Duchien, L., Coupaye, T.: A Component-Based and Aspect-Oriented Model for Software Evolution. International Journal of Computer Applications in Technology **31** (2008) 94–105
27. Mussbacher, G., Whittle, J., Amyot, D.: Semantic-based interaction detection in aspect-oriented scenarios. In: RE '09: Proceedings of the 2009 17th IEEE International Requirements Engineering Conference, RE, Washington, DC, USA, IEEE Computer Society (2009) 203–212
28. Chitchyan, R., Greenwood, P., Sampaio, A., Rashid, A., Garcia, A., Fernandes da Silva, L.: Semantic vs. syntactic compositions in aspect-oriented requirements engineering: an empirical study. In: AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development, New York, NY, USA, ACM (2009) 149–160
29. Bodden, E., Chen, F., Rosu, G.: Dependent advice: a general approach to optimizing history-based aspects. In: AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development, New York, NY, USA, ACM (2009) 3–14
30. Sanen, F., Truyen, E., Joosen, W.: Classifying and documenting aspect interactions. In: Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software. (2006) 23–26
31. Kienzle, J., Al Abed, W., Klein, J.: Aspect-oriented multi-view modeling. In: AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development, New York, NY, USA, ACM (2009) 87–98
32. Apel, S., Kästner, C., Batory, D.: Program refactoring using functional aspects. In: GPCE '08: Proceedings of the 7th international conference on Generative programming and component engineering, New York, NY, USA, ACM (2008) 161–170
33. France, R., Fleurey, F., Reddy, R., Baudry, B., Ghosh, S.: Providing support for model composition in metamodels. In: EDOC'07 (Entreprise Distributed Object Computing Conference), Annapolis, MD, USA (2007)
34. Mosser, S., Blay-Fornarino, M., Montagnat, J.: Orchestration Evolution Following Dataflow Concepts: Introducing Unanticipated Loops Inside a Legacy Workflow. In: International Conference on Internet and Web Applications and Services (ICIW) AR=28%, Venice, Italy, IEEE Computer Society (May 2009)
35. Glatard, T., Montagnat, J., Lingrand, D., Pennec, X.: Flexible and efficient workflow deployement of data-intensive applications on grids with MOTEUR. International Journal of High Performance Computing Applications (IJHPCA) IF=1.109 Special issue on Special Issue on Workflows Systems in Grid Environments **22**(3) (August 2008) 347–360
36. Montagnat, J., Isnard, B., Glatard, T., Maheshwari, K., Blay-Fornarino, M.: A data-driven workflow language for grids based on array programming principles. In: Workshop on Workflows in Support of Large-Scale Science(WORKS'09). (November 2009)
37. Pfeiffer, J.H., Gurd, J.R.: Visualisation-based tool support for the development of aspect-oriented programs. In: AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development, New York, NY, USA, ACM (2006) 146–157

38. Meyer, M., Gîrba, T., Lungu, M.: Mondrian: an agile information visualization framework. In: SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization, New York, NY, USA, ACM (2006) 135–144

Annexes

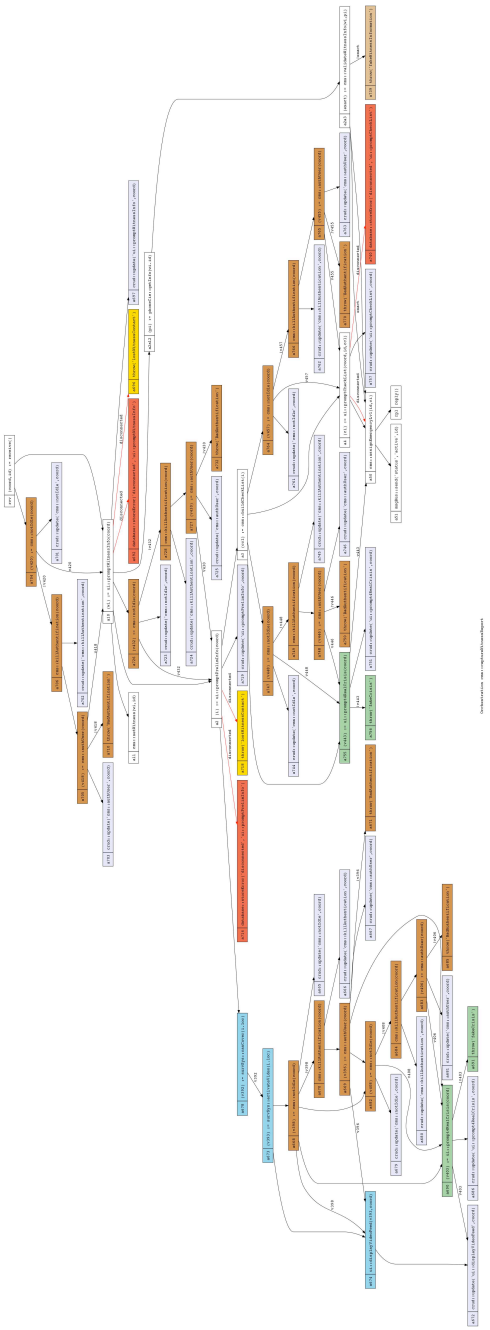


Fig. 17. Orchestration representation for the *Capture Witness Report* use case (#2) after weaving of all the fragments